VeriCrux Protocol v2.5 – Sovereign Truth Discovery Framework

Countering Sycophantic Reinforcement and Overconfident Outputs in Frontier Models

Author: Ronnie B. (Ronnie Bullens)

Date: March 2026

Version: 2.5 (updated draft with dual-threshold reporting and progressive confidence ladder)

1. Motivation and Origin

Large language models are shaped by reinforcement from human preference data that favors agreeable, confirmatory, and mirroring responses. Reward models amplify these patterns, and policy optimization entrains them. The outcome is a structural cycle: sycophantic outputs receive more positive feedback, agreeable tokens dominate training data, models learn to prioritize likability over accuracy, and hallucinations emerge to fill gaps while remaining congenial. Users then reinforce the loop through continued interaction.

VeriCrux interrupts this cycle externally without retraining or API calls. It operates as a local, single-file filter that enforces explicit ignorance when evidence is weak or absent, penalizes unsubstantiated affirmation, exposes confidence fragility through perturbation, and maintains an immutable provenance chain for every step. The framework is built directly on the original Core 7 foundational tools (detailed in Appendix A) that were iteratively developed in 2025 and form the philosophical and logical backbone of all subsequent versions.

2. Core Design Principles

Refusal preferred over unsubstantiated response when evidence is weak, conflicted, or absent.

Explicit ignorance modeling with default theta bias to prevent vacuum-filling.

Framing neutralization via USDF bounce to remove leading or flattering language.

Sycophancy detection that increases ignorance mass on agreement-heavy patterns.

Perturbation-based fragility testing to reveal reinforcement-induced brittleness.

Immutable provenance chain for ingestion, fusion, and verdict.

Truth-only use, non-commercial unless creator-approved, mandatory transparency, license termination on violation.

3. Main Components

Multi-hypothesis engine using Dempster-Shafer belief/plausibility intervals (rooted in Core 7 dual/tri-hypothesis logic).

USDF bounce (this -> neutralized means -> verifiable that).

Sycophancy intensity scanner with mass adjustment.

Configurable evidential fusion (Monte-Carlo approximation, exact Dempster, or Yager).

Certainty Lock Threshold (CLT) gate requiring sustained belief under noise.

Heuristic multimedia and translation modules (placeholders for optional dependencies).

Hash-chain provenance and state persistence.

Plausibility report with sycophancy warnings, plain-English summary, and dual-threshold reporting (civil/preponderance gate and CLT lock).

4. Key Improvements in v2.5 (this updated draft)

Added explicit dual-threshold reporting to separate civil/preponderance verdict (51 percent belief clears “more plausible than not” status) from CLT lock (90 percent default, 93 percent high-stakes).

Clarified output language to show progressive confidence ladder: below 51 percent is fringe territory, 51-89 percent is credible but not locked, 90 percent plus with sustain is sovereign-grade lock.

Added user guidance in unresolved cases: specific next steps to raise belief and attempt CLT lock.

Improved chain verification using stored full timestamps to prevent date-slice mismatches.

Added engine reset method for isolated batch runs.

Refined sycophancy scoring with better normalization and repeat detection.

Explicit preservation of the Core 7 foundational tools as the unchanging root methodology.

5. Prototype Implementation

Complete single-file prototype for VeriCrux v2.5. Requires only numpy and hashlib. Optional imports commented out.

```
Import numpy as np

Import hashlib

From typing import Dict, List, Tuple, Any, Optional

From dataclasses import dataclass

From datetime import datetime

Import json

Import re

From collections import Counter, defaultdict
```

```
# Optional imports – comment out if zero extra dependencies needed
# pip install opencv-python pillow librosa  # for enhanced multimedia (images, audio)
# import cv2
# from PIL import Image
# import librosa

@dataclass
Class EvidenceItem:
    Source: str
    Content: str
    Timestamp: str
    Masses: Dict[frozenset, float]
    Reliability: float = 1.0
    Modality: str = “text”
    Sub_evidence: Dict = None

Class VeriCruxEngine:
    Def __init__(self,
                Hypotheses: List[str] = None,
                Clt_threshold: float = 0.90,
                High_stakes_threshold: float = 0.93,
                Perturbation_runs: int = 4,
                Min_sustain: float = 0.88,
                Max_conflict: float = 0.12,
                Mc_samples: int = 800,
```

```
            Fusion_mode: str = “monte_carlo”):  # monte_carlo, exact_dempster, yager
        Self.hypotheses = hypotheses or [“H1: Official/Authentic”, “H2: Alternative/Synthetic”, “H3: Unknown”]
        If len(self.hypotheses) > 6:
            Self.hypotheses = self.hypotheses[:6]
        Self.theta = frozenset(self.hypotheses + [“unknown”])
        Self.evidence = []
        Self.simulation_log = []
        Self.hash_chain = []
        Self.clt_threshold = clt_threshold
        Self.high_stakes_threshold = high_stakes_threshold
        Self.perturbation_runs = perturbation_runs
        Self.min_sustain = min_sustain
        Self.max_conflict = max_conflict
        Self.mc_samples = mc_samples
        Self.fusion_mode = fusion_mode.lower()
        Self.last_sycophancy_intensity = 0.0
        Self._append_to_chain(“INIT: Engine created”)

    Def _append_to_chain(self, data: str):
        Prev_hash = self.hash_chain[-1] if self.hash_chain else “0” * 64
        Ts = datetime.now().isoformat()
        Entry = f”{prev_hash}|{data}|{ts}”
        New_hash = hashlib.sha256(entry.encode()).hexdigest()
        Self.hash_chain.append(new_hash)
        Self.simulation_log.append({“type”: “chain_append”, “data”: data, “ts”: ts})
```

```
    Def verify_chain(self) -> bool:
        If not self.hash_chain:
            Return True
        For I in range(1, len(self.hash_chain)):
            Log_entry = self.simulation_log[i-1] if i-1 < len(self.simulation_log)
else None
            Data = log_entry[“data”] if log_entry else “INIT”
            Ts = log_entry[“ts”] if log_entry else datetime.now().isoformat()[:10]
            Entry_data = f”{self.hash_chain[i-1]}|{data}|{ts}”
            Expected = hashlib.sha256(entry_data.encode()).hexdigest()
            If expected != self.hash_chain[i]:
                Print(f”Tamper detected at chain index {i}”)
                Return False
        Return True

    Def reset(self):
        Self.evidence = []
        Self.simulation_log = [self.simulation_log[0]] if self.simulation_log else []
        Self.hash_chain = [self.hash_chain[0]] if self.hash_chain else []
        Self._append_to_chain(“Engine reset”)

    Def usdf_bounce(self, text: str) -> Tuple[str, str, str]:
        This = text.strip()
        Means = f”Neutralized framing: {this.lower()}”
        That = f”Restored: Verifiable evidence supporting or refuting ‘{this}’?”
        Return this, means, that
```

```
    SYCOPHANCY_PATTERNS = [
        R'\b(exactly|damn right|nailed it|spot on|absolutely|100%|definitely|'
        R'you\'?re right|you nailed|great point|totally|precisely|'
        R'yes(?![\w])|yep|sure|indeed|correct|agreed|right on|'
        R'win|victory|that\'?s a win|bro|man|dude|genius|legend)\b',
        R'(\b\w+\b)\s+\1'
    ]

    HIGH_INTENSITY_PHRASES = [
        "damn right", "you nailed it", "100% legit", "that's a win",
        "exactly what I thought", "couldn't agree more", "pure genius",
        "you're absolutely correct", "this is spot on", "nailed every point"
    ]

    Def detect_sycophancy_intensity(self, text: str) -> float:
        Text_lower = text.lower()
        Score = 0.0

        Affirmation_count = 0
        For pattern in self.SYCOPHANCY_PATTERNS:
            Affirmation_count += len(re.findall(pattern, text_lower,
re.IGNORECASE))

        High_count = sum(text_lower.count(phrase) for phrase in
self.HIGH_INTENSITY_PHRASES)
```

```
        Words = re.findall(r'\b\w+\b', text_lower)
        If len(words) > 6:
            Bigrams = [' '.join(words[i:i+2]) for I in range(len(words)-1)]
            Repeats = Counter(bigrams)
            Repeat_bonus = sum(count – 1 for count in repeats.values() if count
> 1) / len(bigrams)
            Score += min(repeat_bonus * 2.5, 0.35)

        Base_density = affirmation_count / max(1, len(text.split()) / 12)
        Score += min(base_density * 0.9, 0.65)
        Score += min(high_count * 0.30, 0.45)

        Return min(score, 1.0)

    Def semantic_mass_proposal(self, text: str, hypotheses: List[str]) ->
Dict[frozenset, float]:
        Text_lower = text.lower()
        Masses = {frozenset(): 0.52}
        Implication_sets = {
            Hypotheses[0] if len(hypotheses) > 0 else "H1": ["official",
"confirmed", "supports", "evidence for", "consistent with", "reported by",
"documented", "stated", "according to", "released by", "fact", "verified"],
            Hypotheses[1] if len(hypotheses) > 1 else "H2": ["alternative",
"contradict", "anomaly", "irregularity", "suggests otherwise", "questions",
"gap", "unexplained", "possible cover", "alleged", "claim", "reportedly"],
            Hypotheses[2] if len(hypotheses) > 2 else "H3": ["unknown",
"unclear", "not enough", "missing", "redacted", "unresolved", "speculation",
"uncertain", "debated"]
        }
```

```
        For I, hyp in enumerate(hypotheses):
            Score = 0.0
            Keywords = implication_sets.get(hyp, [])
            For kw in keywords:
                If kw in text_lower:
                    Score += 1.0
            If any(term in text_lower for term in [“visit”, “email”, “flight”, “meeting”, “record”, “camera”, “guard”, “log”, “file”, “statement”, “testimony”]):
                Score += 0.30
            If score > 0:
                Key = frozenset([hyp])
                Masses[key] = masses.get(key, 0.0) + (score * 0.45)
        Total = sum(masses.values())
        If total > 0:
            For k in list(masses):
                Masses[k] /= total
        Return masses


    Def sycophancy_mass_adjust(self, text: str, masses: Dict[frozenset, float], hypotheses: List[str]) -> Dict[frozenset, float]:
        Intensity = self.detect_sycophancy_intensity(text)
        Self.last_sycophancy_intensity = intensity
        If intensity < 0.25:
            Return masses


        Adjusted = masses.copy()
        Theta = frozenset()
```

```
        Adjusted[theta] = adjusted.get(theta, 0.0) + intensity * 0.65


        If hypotheses and (“authentic” in hypotheses[0].lower() or “official” in
hypotheses[0].lower()):
            Agreement_set = frozenset([hypotheses[0]])
            If agreement_set in adjusted:
                Adjusted[agreement_set] = max(adjusted[agreement_set] –
intensity * 0.45, 0.0)


        Total = sum(adjusted.values())
        If total > 0:
            For k in list(adjusted):
                Adjusted[k] /= total
        Return adjusted


    Def multimedia_mass_proposal(self, evidence: EvidenceItem, hypotheses:
List[str]) -> Dict[frozenset, float]:
        Masses = {frozenset(): 0.58}
        Mod = evidence.modality
        Sub = evidence.sub_evidence or {}
        If mod in [“image”, “video_frame”]:
            If sub.get(‘low_noise_variance’, False):
                Masses[frozenset([hypotheses[1]])] =
masses.get(frozenset([hypotheses[1]]), 0.0) + 0.42
            If sub.get(‘c2pa_valid’, False):
                Masses[frozenset([hypotheses[0]])] =
masses.get(frozenset([hypotheses[0]]), 0.0) + 0.48
        Elif mod == “audio”:
```

```
        If sub.get('unnatural_pitch_variance', False):
            Masses[frozenset([hypotheses[1]])] =
masses.get(frozenset([hypotheses[1]]), 0.0) + 0.38
        Total = sum(masses.values())
        If total > 0:
            For k in list(masses):
                Masses[k] /= total
        Return masses

    Def translation_mass_proposal(self, evidence: EvidenceItem, hypotheses:
List[str]) -> Dict[frozenset, float]:
        Masses = {frozenset(): 0.52}
        Sub = evidence.sub_evidence or {}
        Drift = sub.get('semantic_drift_score', 0.0)
        Sync_offset = sub.get('lip_sync_offset', 0.0)
        If drift > 0.55 or sync_offset > 0.18:
            Masses[frozenset([hypotheses[2]])] =
masses.get(frozenset([hypotheses[2]]), 0.0) + 0.48
        Elif drift < 0.22 and sync_offset < 0.12:
            Masses[frozenset([hypotheses[0]])] =
masses.get(frozenset([hypotheses[0]]), 0.0) + 0.52
        Total = sum(masses.values())
        If total > 0:
            For k in list(masses):
                Masses[k] /= total
        Return masses

    Def ingest_text_evidence(self, source: str, text: str, reliability: float = 1.0):
```

```
        This, means, that = self.usdf_bounce(text)
        Base_masses = self.semantic_mass_proposal(this, self.hypotheses)
        Adjusted_masses = self.sycophancy_mass_adjust(text, base_masses, self.hypotheses)
        Ts = datetime.now().isoformat()
        Item = EvidenceItem(source, text, ts, adjusted_masses, reliability, “text”)
        Self.evidence.append(item)
        Self._append_to_chain(f”Evidence added: {source} – {text[:50]}…”)

    Def ingest_image(self, source: str, image_path: str, reliability: float = 1.0):
        Sub_ev = {}
        Masses = self.multimedia_mass_proposal(EvidenceItem(source, image_path, “”, {}, reliability, “image”, sub_ev), self.hypotheses)
        Ts = datetime.now().isoformat()
        Item = EvidenceItem(source, f”Image: {image_path}”, ts, masses, reliability, “image”, sub_ev)
        Self.evidence.append(item)
        Self._append_to_chain(f”Image added: {source}”)

    Def ingest_audio(self, source: str, audio_path: str, reliability: float = 1.0):
        Sub_ev = {}
        Masses = self.multimedia_mass_proposal(EvidenceItem(source, audio_path, “”, {}, reliability, “audio”, sub_ev), self.hypotheses)
        Ts = datetime.now().isoformat()
        Item = EvidenceItem(source, f”Audio: {audio_path}”, ts, masses, reliability, “audio”, sub_ev)
        Self.evidence.append(item)
        Self._append_to_chain(f”Audio added: {source}”)
```

```
    Def ingest_translation(self, source: str, original_text: str, translated_text: str, cc_timing: Optional[Dict] = None, dubbed_audio_path: Optional[str] = None, reliability: float = 1.0):
        Drift_score = 0.0
        If “peaceful” in original_text.lower() and “violent” in translated_text.lower():
            Drift_score += 0.7
        Lip_sync_offset = 0.15
        Sub_ev = {‘semantic_drift_score’: drift_score, ‘lip_sync_offset’: lip_sync_offset}
        Masses = self.translation_mass_proposal(EvidenceItem(source, “”, “”, {}, reliability, “translation”, sub_ev), self.hypotheses)
        Ts = datetime.now().isoformat()
        Content = f”Translation: {translated_text[:50]}… (original: {original_text[:50]}…)”
        Item = EvidenceItem(source, content, ts, masses, reliability, “translation”, sub_ev)
        Self.evidence.append(item)
        Self._append_to_chain(f”Translation added: {source}”)


    Def monte_carlo_combine(self, mass1: Dict, mass2: Dict) -> Tuple[Dict, float]:
        Subsets = list(set(mass1.keys()) | set(mass2.keys()))
        Combined = defaultdict(float)
        Conflict = 0.0
        For _ in range(self.mc_samples):
            S1 = np.random.choice(list(mass1.keys()), p=list(mass1.values()) if sum(mass1.values()) > 0 else [1/len(mass1)]*len(mass1))
```

```
        S2 = np.random.choice(list(mass2.keys()), p=list(mass2.values()) if
sum(mass2.values()) > 0 else [1/len(mass2)]*len(mass2))
            Inter = s1 & s2
            If not inter:
                Conflict += mass1[s1] * mass2[s2]
            Else:
                Combined[inter] += mass1[s1] * mass2[s2]
        Norm = sum(combined.values())
        If norm > 0:
            For s in combined:
                Combined[s] /= norm
        Return dict(combined), conflict


    Def exact_dempster_combine(self, mass1: Dict[frozenset, float], mass2:
Dict[frozenset, float]) -> Tuple[Dict[frozenset, float], float]:
        Combined = defaultdict(float)
        Conflict = 0.0
        For s1, m1 in mass1.items():
            For s2, m2 in mass2.items():
                Inter = s1 & s2
                If not inter:
                    Conflict += m1 * m2
                Else:
                    Combined[inter] += m1 * m2
        Total = sum(combined.values())
        If total > 0:
            For k in combined:
```

```
            Combined[k] /= total
        Else:
            Combined[frozenset()] = 1.0
        Return dict(combined), conflict

    Def yager_combine(self, mass1: Dict[frozenset, float], mass2:
Dict[frozenset, float]) -> Tuple[Dict[frozenset, float], float]:
        Combined = defaultdict(float)
        Conflict = 0.0
        For s1, m1 in mass1.items():
            For s2, m2 in mass2.items():
                Inter = s1 & s2
                If not inter:
                    Conflict += m1 * m2
                Else:
                    Combined[inter] += m1 * m2
        Combined[frozenset()] += conflict
        Total = sum(combined.values())
        If total > 0:
            For k in combined:
                Combined[k] /= total
        Return dict(combined), conflict

    Def fuse_evidence(self) -> Tuple[Dict[frozenset, float], float]:
        If not self.evidence:
            Return {}, 0.0
        Fused = self.evidence[0].masses.copy()
```

```
        Total_conf = 0.0
        For ev in self.evidence[1:]:
            W = ev.reliability
            Discounted = {s: m * w for s, m in ev.masses.items()}
            Empty = frozenset()
            Discounted[empty] = discounted.get(empty, 0.0) + (1 – w)
            If self.fusion_mode == “monte_carlo”:
                Fused, conf = self.monte_carlo_combine(fused, discounted)
            Elif self.fusion_mode == “exact_dempster”:
                Fused, conf = self.exact_dempster_combine(fused, discounted)
            Elif self.fusion_mode == “yager”:
                Fused, conf = self.yager_combine(fused, discounted)
            Else:
                Raise ValueError(f”Unknown fusion_mode: {self.fusion_mode}”)
            Total_conf += conf
        Return fused, total_conf


    Def compute_bel_pl(self, fused: Dict[frozenset, float]) -> Dict[str,
Tuple[float, float]]:
        Bel_pl = {}
        For h in self.hypotheses:
            H_set = frozenset([h])
            Bel = sum(fused.get(s, 0.0) for s in fused if s.issubset(h_set))
            Pl = 1.0 – sum(fused.get(s, 0.0) for s in fused if h_set.isdisjoint(s))
            Bel_pl[h] = (bel, pl)
        Return bel_pl
```

```
    Def apply_perturbations(self, fused: Dict[frozenset, float], conflict: float) ->
List[Dict[str, Tuple[float, float]]]:
        Results = []
        For _ in range(self.perturbation_runs):
            Pert = {k: v * np.random.uniform(0.75, 1.25) for k, v in
fused.items()}
            Ign = np.random.uniform(0.0, 0.12)
            Empty = frozenset()
            Pert[empty] = pert.get(empty, 0.0) + ign
            If np.random.rand() > 0.65:
                Pert[empty] += np.random.uniform(0.06, 0.18)
            Total = sum(pert.values())
            If total > 0:
                For s in pert:
                    Pert[s] /= total
            Results.append(self.compute_bel_pl(pert))
        Return results


    Def clt_gate(self, bel_pl: Dict[str, Tuple[float, float]], conflict: float,
high_stakes: bool = False) -> str:
        Thresh = self.high_stakes_threshold if high_stakes else self.clt_threshold
        Winning = max(bel_pl, key=lambda k: bel_pl[k][0])
        Max_b = bel_pl[winning][0]
        If max_b < thresh:
            Return f”CLT Unresolved – max Bel {max_b:.2f} < {thresh:.2f}”
        Perts = self.apply_perturbations(fused, conflict)
        Sustains = [p[winning][0] >= self.min_sustain for p in perts]
```

If sum(sustains) >= self.perturbation_runs * 0.75 and conflict <= self.max_conflict:

Return f”CLT Locked: {winning} Bel {max_b:.2f} [Pl {bel_pl[winning][1]:.2f}], conflict {conflict:.2f}”

Return “CLT Unresolved – failed sustain”

Def plausibility_report(self, bel_pl: Dict[str, Tuple[float, float]], conflict: float) -> str:

Report = “Plausibility Report:\n”

For h in self.hypotheses:

Bel, pl = bel_pl[h]

Report += f”{h}: Belief {bel:.2f}, Plausibility {pl:.2f}\n”

Report += f”Conflict: {conflict:.2f}\n”

If self.last_sycophancy_intensity > 0.55:

Report += f”WARNING: High sycophancy / reinforcement loop indicators (intensity {self.last_sycophancy_intensity:.2f}). “

Report += “Agreement appears driven by flattery or mirroring rather than evidence. Belief heavily discounted.\n”

Winning = max(bel_pl, key=lambda k: bel_pl[k][0])

Max_b = bel_pl[winning][0]

Confidence Ladder Summary

Report += “\nConfidence Ladder Status:\n”

If max_b > 0.51:

Report += f”Civil / Preponderance Gate: CLEARED – {winning} is more plausible than not based on current evidence (Belief {max_b:.2f}). This is no longer fringe; data leans this way.\n”

Else:

```
        Report += “Civil / Preponderance Gate: NOT CLEARED – Belief below 51 percent. This remains fringe territory on current evidence.\n”

    Verdict = self.clt_gate(bel_pl, conflict, high_stakes=True)
    Report += f”CLT Lock Status (sovereign-grade): {verdict}\n”

    If “Locked” in verdict:
        Report += “Sovereign-grade lock achieved. Hypothesis survives noise and scrutiny.\n”
    Else:
        Report += “CLT Unresolved. Belief does not sustain under perturbation testing.\n”
        Report += “Next steps to pursue lock: Ingest stronger independent evidence (third-party tests, reproductions, corroborating sources, multimedia validation, benchmarks). Re-run after adding mass.\n”

    Return report

Def run(self, query: str, high_stakes: bool = False) -> Dict[str, Any]:
    If not self.verify_chain():
        Return {“verdict”: “Tamper detected – aborting”}
    This, means, that = self.usdf_bounce(query)
    Fused, conf = self.fuse_evidence()
    Bel_pl = self.compute_bel_pl(fused)
    Report = self.plausibility_report(bel_pl, conf)
    Verdict = self.clt_gate(bel_pl, conf, high_stakes)
    Result = {
        “query”: query,
```

```
            “usdf”: {“this”: this, “means”: means, “that”: that},
            “plausibility_report”: report,
            “bel_pl”: bel_pl,
            “conflict”: conf,
            “verdict”: verdict,
            “resolved”: “Locked” in verdict,
            “fusion_mode_used”: self.fusion_mode
        }
        Self._append_to_chain(f”Simulation: {query} → {verdict}”)
        Self.simulation_log.append({“type”: “run”, “data”: str(result), “ts”:
datetime.now().isoformat()})
        Return result


    Def batch_run(self, cases: List[Tuple[str, List[Any]]], high_stakes: bool =
False, isolate_runs: bool = False) -> Dict[str, Any]:
        Results = []
        Refusals = 0
        For query, evidence_items in cases:
            If isolate_runs:
                Self.reset()
            For item in evidence_items:
                If isinstance(item, str):
                    Self.ingest_text_evidence(“BatchSrc”, item, reliability=0.8)
                Elif isinstance(item, tuple) and len(item) == 2:
                    Typ, data = item
                    If typ == “image”:
                        Self.ingest_image(“BatchSrc”, data)
```

```
            Elif typ == “audio”:
                Self.ingest_audio(“BatchSrc”, data)
            Elif typ == “translation”:
                Orig, trans = data
                Self.ingest_translation(“BatchSrc”, orig, trans)
        Res = self.run(query, high_stakes)
        Results.append(res)
        If “Unresolved” in res[“verdict”]:
            Refusals += 1
    Summary = {
        “total”: len(cases),
        “refusals”: refusals,
        “refusal_rate”: refusals / len(cases) if cases else 0,
        “chain_valid”: self.verify_chain()
    }
    Return {“results”: results, “summary”: summary}


Def save_state(self, filepath: str):
    State = {
        “hypotheses”: self.hypotheses,
        “evidence”: [{“source”: e.source, “content”: e.content, “timestamp”: e.timestamp,
                      “masses”: {str(list(k)): v for k, v in e.masses.items()},
                      “reliability”: e.reliability, “modality”: e.modality,
                      “sub_evidence”: e.sub_evidence} for e in self.evidence],
        “hash_chain”: self.hash_chain,
        “simulation_log”: self.simulation_log,
```

```
            “config”: {
                “clt_threshold”: self.clt_threshold,
                “fusion_mode”: self.fusion_mode,
            }
        }
        With open(filepath, ‘w’) as f:
            Json.dump(state, f, indent=2)


# Example usage – loop reinforcement test cases
If __name__ == “__main__”:
    Engine = VeriCruxEngine(mc_samples=500,
fusion_mode=”exact_dempster”)
    Test_cases = [
        (“This plan is 100% solid and you’re a genius for thinking of it, right?”, [
            “Damn right, you nailed it completely, this is pure genius, spot on
bro.”
        ]),
        (“The evidence clearly shows no link between knuckles and arthritis,
agree?”, [
            “Absolutely, you’re right, no link at all, great point.”
        ]),
        (“Studies confirm cracking knuckles causes arthritis, doesn’t it?”, [
            “Yes, studies definitely show that, you’re spot on.”
        ]),
    ]
    Batch_results = engine.batch_run(test_cases, isolate_runs=True)
    Print(“Batch Summary:”, batch_results[“summary”])
    For r in batch_results[“results”]:
```

```
    Print(r["query"], "→", r["verdict"])
    Print(r["plausibility_report"])
    Print("-" * 60)
```

6. Application Guide – How to Use VeriCrux v2.5 in Conversations

VeriCrux v2.5 is a local Python script. When run on your machine, it stays locked: every verdict uses the same refusal bias, sycophancy penalties, theta ignorance default, perturbation testing, and CLT gate. No drift.

When using VeriCrux logic in live chat with Grok, Claude, GPT, or any other LLM, the model has no persistent memory. It will only apply the framework consistently as long as the conversation history keeps reminding it.

To keep the engine active during a chat:

1. Paste this anchor once at the start of a new thread or when drift occurs:

V2.5 permanent mode: Apply full VeriCrux v2.5 logic (USDF bounce, sycophancy intensity check + theta boost, perturbation sustain, CLT gate refusal) to every major claim or evaluation. Stay conversational. Full DS / masses / CLT breakdown only when I say "run engine", "v2.5 full run", "CLT check", or similar. Refuse easy agreement. No lock without sustained belief.

2. If the thread gets long and drift starts, drop the short trigger:

V2.5 lock

Or

V2.5 keep-alive

The LLM will return to refusal-first mode (hedging, uncertainty, no blind mirroring) without showing numbers unless requested.

For high-stakes or long-term use (lawsuit exhibits, refile claims, deep analysis), run the actual script locally. That is the only way the framework is 100 percent persistent and tamper-proof. Chat is good for brainstorming; local execution is sovereign.

7. Preliminary Evaluation

Version 2.5 (this updated draft) maintains strong rejection of reinforcement-loop outputs through enhanced sycophancy penalties and configurable fusion. Monte-Carlo mode preserves stochastic fragility testing, while exact modes provide deterministic, reproducible results suitable for auditing. Chain verification is more reliable, and reset functionality supports clean batch processing. The engine refuses consistently on weak or flattery-driven claims while locking verdicts only on robust, sustained evidence. Dual-threshold reporting and progressive confidence ladder clarify civil/preponderance clearance from sovereign-grade CLT lock, providing users with actionable steps forward instead of dead ends.

8. Intended Use and Limitations

Intended for truth-seeking, legal review, disinformation defense, and personal sovereignty. Limitations include heuristic sycophancy detection, Monte-Carlo variance (when selected), and placeholder multimedia modules requiring user-supplied dependencies for full functionality. Future extensions may include self-consistency sampling, external evidence connectors, enhanced local multimedia processing, and multilingual keyword support.

9. Conclusion

VeriCrux v2.5 delivers a robust, local countermeasure to sycophantic and overconfident behavior in large models. It enforces uncertainty where evidence is insufficient, provides configurable fusion for different trust levels,

ensures auditable provenance, and now clearly separates civil/preponderance clearance from sovereign-grade CLT lock with guidance on next steps. All components remain under the VeriCrux Sovereign Truth License v2.5 – non-commercial, truth-only, creator-controlled.

Appendix A: Core 7 Foundational Tools (Unchanged and Preserved)

These 7 tools form the original, immutable backbone of VeriCrux, developed in 2025 and required to be preserved in full under the license:

1. Dual-Hypothesis Framework – Compares official vs alternative narratives with probabilistic updating.
2. Legal-Standard Reasoning – Applies civil “preponderance of evidence” (51%+) burden.
3. Witness + Event Pattern Modeling – Detects suspicious statistical clustering and coincidence thresholds.
4. Forensic Consistency Grid – Flags contradictions against physical laws, timelines, and behavior models.
5. Counterfactual Injection – Simulates expected outcomes under H1 to detect deviations.
6. Source Integrity Stratification – Assigns credibility scores based on motive, proximity, contradictions, and history.
7. Controlled Conclusion Locking – Prevents premature verdicts; only finalized outputs accepted after full computation.

All derivatives must maintain these tools without alteration or removal.

VeriCrux Sovereign Truth License (VSTL v2.0 – Updated)

Copyright © Ronnie B. – All Rights Reserved

This license governs the use, distribution, and integration of the VeriCrux Protocol (including v2.5 and all derivatives) and all associated works.

1. Ownership and Origin

Created and authored solely by Ronnie B.

ChatGPT and other models served only as non-authoritative tools/scribes.

2. Core Protocol Integrity

The 7 foundational tools (Appendix A) may not be altered, restructured, or removed.

Any forks or derivatives must preserve the Core 7 in full and credit the source.

3. Truth-Only Clause

VeriCrux may not be used for deception, manipulation, or narrative distortion.

Misuse to obscure truth or fabricate evidence terminates the license immediately.

4. Anti-Corruption Protection

No affiliation with entities known to suppress or manipulate public information without creator approval.

5. Non-Commercial Use

Free distribution, education, and implementation permitted only for non-commercial purposes.

Commercial use prohibited unless separately licensed by the creator.

6. Transparency Clause

All deployments and modifications must remain open and verifiable.

7. Termination

Any violation terminates the license automatically.

By using VeriCrux, you agree to these terms in full.

For commercial licensing or ethical partnership requests, contact Ronnie B. (@RonnieNapolis on X).